Client-Side Slice-to-C++ Mapping

Client-Side Slice-to-C++ Mapping

- The client-side Slice-to-C++ mapping defines how Slice data types are translated to C++ types, and how clients invoke operations, pass parameters, and handle errors.
- The mapping is free from the potential pitfalls of memory management: all types are self-managed and automatically clean up when instances go out of scope.
  - You cannot accidentally introduce a memory leak by:
    - Ignoring the return value of an operation invocation
    - Forgetting to deallocate memory that was allocated by a called operation.
- The C++ mapping is fully thread-safe.
  - You must still synchronize access to data from different threads

Mapping for Identifiers

- Slice identifiers map to an identical C++ identifier
  - The Slice identifier **`Clock`** becomes the C++ identifier **`Clock`**.
- If a Slice identifier is the same as a C++ keyword, the corresponding C++ identifier is prefixed with **`_cpp_`**. For example, the Slice identifier **`while`** is mapped as **`_cpp_while`**.
- A single Slice identifier often results in several C++ identifiers.
  - For a Slice interface named **`Foo`**, the generated C++ code uses the identifiers **`Foo`** and **`FooPrx`** (among others).
  - If the interface has the name while, the generated identifiers are **`_cpp_while`** and **`whilePrx`** (not **`_cpp_whilePrx`**).
    - Te prefix is applied only to those generated identifiers that actually require it.

Mapping for Modules

- Slice modules map to C++ namespaces. The mapping preserves the nesting of the Slice definitions.

```
module M1 {
  module M2 {
    // ...
  };
  // ...
};
// ...
module M1 { // Reopen M1
  // ...
};
```

- This definition maps to the corresponding C++ definition:

```
namespace M1 {
  namespace M2 {
    // ...
  }
  // ...
}
// ...
namespace M1 { // Reopen M1
  // ...
}
```

- If a Slice module is reopened, the corresponding C++ namespace is reopened as well.

Mapping for Simple Built-In Types

Slice	C++
bool	bool
byte	Ice::Byte
short	Ice::Short
int	Ice::Int
long	Ice::Long
float	Ice::Float
double	Ice::Double
string	std::string

- Slice **`bool`** and **`string`** map to C++ **`bool`** and **`std::string`**.
- The remaining built-in Slice types map to C++ type definitions instead of C++ native types.
  - This allows the Ice run time to provide a definition as appropriate for each target architecture.
    - **`Ice::Int`** might be defined as **`long`** on one architecture and as **`int`** on another.)
- **`Ice::Byte`** is a typedef for **`unsigned char`**.
  - This guarantees that byte values are always in the range 0..255.
- All the basic types are guaranteed to be distinct C++ types, that is, you can safely overload functions that differ in only the types in the table above

Mapping for Enumerations

- Enumerations map to the corresponding enumerations in C++:
- The Slice definition:
  - `enum Fruit { Apple, Pear, Orange };`
- maps to:
  - `enum Fruit { Apple, Pear, Orange };`

Mapping for Structures

- By default, Slice structures map to C++ structures with the same name. For each Slice data member, the C++ structure contains a public data member.

```
struct Employee {
  long number;
  string firstName;
  string lastName;
};
```

- The Slice-to-C++ compiler generates the following definition for this structure:

```
struct Employee {
  Ice::Long number;
  std::string firstName;
  std::string lastName;
  bool operator==(const Employee&) const;
  bool operator!=(const Employee&) const;
  bool operator<(const Employee&) const;
  bool operator<=(const Employee&) const;
  bool operator>(const Employee&) const;
  bool operator>=(const Employee&) const;
};
```

Mapping for Structures

- The comparison operators treat the members of a structure as sort order criteria: the first member is considered the first criterion, the second member the second criterion, and so on.
- The comparison operators are provided to allow the use of structures as the key type of Slice dictionaries.
- Copy construction and assignment always have deep-copy semantics.
  - You can freely assign structures or structure members to each other without having to worry about memory management.

Mapping for Sequences

- The sequences map by default to STL vectors. From the Slice definition:
  - **`sequence<Fruit> FruitPlatter;`**
- The Slice compiler generates the following C++ definition:
  - **`typedef std::vector<Fruit> FruitPlatter;`**

Mapping for Dictionaries

- A Slice dictionary maps to an STL map. Slice code:
  - **`dictionary<long, Employee> EmployeeMap;`**
- will result in the following C++ code being generated:
  - **`typedef std::map<Ice::Long, Employee> EmployeeMap;`**

Mapping for Constants

- Slice constant definitions map to corresponding C++ constant definitions:

```
const bool AppendByDefault = true;
const byte LowerNibble = 0x0f;
const string Advice = "Don't Panic!";
const short TheAnswer = 42;
const double PI = 3.1416;
enum Fruit { Apple, Pear, Orange };
const Fruit FavoriteFruit = Pear;
```

- will map to:

```
const bool AppendByDefault = true;
const Ice::Byte LowerNibble = 15;
const std::string Advice = "Don't Panic!";
const Ice::Short TheAnswer = 42;
const Ice::Double PI = 3.1416;
enum Fruit { Apple, Pear, Orange };
const Fruit FavoriteFruit = Pear;
```

- All constants are initialized directly in the header file, so they are compile-time constants and can be used in contexts where a compile-time constant expression is required.

Mapping for Exceptions

- Each Slice exception is mapped to a C++ class with the same name.

```
exception GenericError {
  string reason;
};
exception BadTimeVal extends GenericError {};
```

- These exception definitions map as follows:

```
class GenericError: public Ice::UserException {
  public:
  std::string reason;
  GenericError() {}
  explicit GenericError(const string&);
  virtual const std::string& ice_name() const;
  virtual Ice::Exception* ice_clone() const;
  virtual void ice_throw() const;
  // Other member functions here...
};
class BadTimeVal: public GenericError {
public:
  BadTimeVal() {}
  explicit BadTimeVal(const string&);
  virtual const std::string& ice_name() const;
  virtual Ice::Exception* ice_clone() const;
  virtual void ice_throw() const;
  // Other member functions here...
};
```

Mapping for Exceptions

- For each exception member, the corresponding class contains a public data member.
- The inheritance structure of the Slice exceptions is preserved for the generated classes, so **`BadTimeVal`** inherits from **`GenericError`**.
- Each exception has three additional member functions:
  - **`ice_name`**
    - Returns the name of the exception.
  - **`ice_clone`**
    - Allows you to polymorphically clone an exception.
  - **`ice_throw`**
    - Allows you to throw an exception without knowing its precise run-time type.

Mapping for Exceptions

- Each exception has a default constructor. This constructor performs memberwise initialization;
  - for simple built-in types, such as integers, the constructor performs no initialization,
  - complex types, such as strings, sequences, and dictionaries are initialized by their respective default constructors.
- An exception also has a second constructor that accepts one argument for each exception member.
  - This constructor allows you to instantiate and initialize an exception in a single statement, instead of having to first instantiate the exception and then assign to its members.
  - For derived exceptions, the constructor accepts one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.
- All user exceptions ultimately inherit from **`Ice::UserException`**. In turn, **`Ice::UserException`** inherits from **`Ice::Exception`** (which is an alias for **`IceUtil::Exception`**).

Mapping for Run-Time Exceptions

- The Ice run time throws run-time exceptions for a number of pre-defined error conditions.
  - All run-time exceptions directly or indirectly derive from **`Ice::LocalException`** (which, in turn, derives from **`Ice::Exception`**).
  - **`Ice::LocalException`** has the usual member functions (**`ice_name`**, **`ice_clone`**, **`ice_throw`**, and (inherited from **`Ice::Exception`**), **`ice_print`**, **`ice_file`**, and **`ice_line`**).
- **`Ice::Exception`**
  - This is the root of the complete inheritance tree. Catching **`Ice::Exception`** catches both user and run-time exceptions.
  - **`Ice::UserException`** This is the root exception for all user exceptions. Catching **`Ice::UserException`** catches all user exceptions (but not run-time exceptions).
- **`Ice::LocalException`** This is the root exception for all run-time exceptions. Catching **`Ice::LocalException`** catches all run-time exceptions (but not user exceptions).
- **`Ice::TimeoutException`**
  - This is the base exception for both operation-invocation and connection-establishment timeouts.
- **`Ice::ConnectTimeoutException`**
  - This exception is raised when the initial attempt to establish a connection to a server times out.

Mapping for Interfaces

- The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that represents the remote object.
- This makes the mapping easy and intuitive to use because, for all intents and purposes (apart from error semantics), making a remote procedure call is no different from making a local procedure call.

Proxy Classes and Proxy Handles

- On the client side, interfaces map to classes with member functions that correspond to the operations on those interfaces. From the following interface:

```
module M {
  interface Simple {
    void op();
  };
};
```

- The Slice compiler generates the following definitions for use by the client:

```
namespace IceProxy {
  namespace M {
    class Simple;
  }
}
namespace M {
  class Simple;
  typedef IceInternal::ProxyHandle< ::IceProxy::M::Simple> SimplePrx;
  typedef IceInternal::Handle< ::M::Simple> SimplePtr;
}
namespace IceProxy {
  namespace M {
    class Simple : public virtual IceProxy::Ice::Object {
    public:
      typedef ::M::SimplePrx ProxyType;
      typedef ::M::SimplePtr PointerType;
      void op();
      void op(const Ice::Context&);
      // ...
    };
  };
}
```

Proxy Classes and Proxy Handles

- In the client's address space, an instance of **`IceProxy::M::Simple`** is the local ambassador for a remote instance of the **`Simple`** interface in a server and is known as a proxy class instance.
  - All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.
- **`Simple`** inherits from **`IceProxy::Ice::Object`**. This reflects the fact that all Ice interfaces implicitly inherit from **`Ice::Object`**.
- For each operation in the interface, the proxy class has two overloaded member functions of the same name.
  - The operation **`op`** has been mapped to two member functions **`op`**.
    - One of the overloaded member functions has a trailing parameter of type **`Ice::Context`**.
      - This parameter is for use by the Ice run time to store information about how to deliver a request; normally, you do not need to supply a value here and can pretend that the trailing parameter does not exist.

Proxy Classes and Proxy Handles

- Client-side application code never manipulates proxy class instances directly.
  - You are not allowed to instantiate a proxy class directly.
- Proxy instances are always instantiated on behalf of the client by the Ice run time.
- When the client receives a proxy from the run time, it is given a proxy handle to the proxy, of type **`<interface-name>Prx`** (**`SimplePrx`** for the preceding example).
- The client accesses the proxy via its proxy handle; the handle takes care of forwarding operation invocations to its underlying proxy, as well as reference-counting the proxy.
  - No memory-management issues can arise: deallocation of a proxy is automatic and happens once the last handle to the proxy disappears (goes out of scope).

Methods on Proxy Handles

- The handle is actually a template of type **`IceInternal::ProxyHandle`** that takes the proxy class as the template parameter. This template has the usual constructor, copy constructor, and assignment operator.
- Default constructor
  - You can default-construct a proxy handle. The default constructor creates a proxy that points nowhere (that is, points at no object at all). If you invoke an operation on such a null proxy, you get an **`IceUtil::NullHandleException`**:

```
try {
  SimplePrx s; // Default-constructed proxy
  s->op(); // Call via nil proxy
  assert(0); // Can't get here
} catch (const IceUtil::NullHandleException&) {
  cout << "As expected, got a NullHandleException"
     << endl;
}
```

Methods on Proxy Handles

- Copy constructor
  - The copy constructor ensures that you can construct a proxy handle from another proxy handle. Internally, this increments a reference count on the proxy; the destructor decrements the reference count again and, once the count drops to zero, deallocates the underlying proxy class instance. That way, memory leaks are avoided:

```
{ // Enter new scope
  SimplePrx s1 = ...; // Get a proxy from somewhere
  SimplePrx s2(s1);   // Copy-construct s2
  assert(s1 == s2);   // Assertion passes
}                     // Leave scope; s1, s2, and the
                      // underlying proxy instance
                      // are deallocated
```

Methods on Proxy Handles

- Assignment operator
  - You can freely assign proxy handles to each other. The handle implementation ensures that the appropriate memory-management activities take place. Self-assignment is safe and you do not have to guard against it:

```
SimplePrx s1 = ...; // Get a proxy from somewhere
SimplePrx s2;       // s2 is nil
s2 = s1;            // both point at the same object
s1 = 0;             // s1 is nil
s2 = 0;             // s2 is nil
```

  - Widening assignments work implicitly. For example, if we have two interfaces, **`Base`** and **`Derived`**, we can widen a **`DerivedPrx`** to a **`BasePrx`** implicitly:

```
BasePrx base;

DerivedPrx derived;

base = derived; // Fine, no problem

derived = base; // Compile-time error
```

  - Implicit narrowing conversions result in a compile error, so the usual C++ semantics are preserved: you can always assign a derived type to a base type, but not vice versa.

Methods on Proxy Handles

- Checked cast
  - Proxy handles provide a **`checkedCast`** method:

```
namespace IceInternal {
  template<typename T>
    class ProxyHandle : public IceUtil::HandleBase<T> {
    public:
    template<class Y>
      static ProxyHandle checkedCast(const ProxyHandle<Y>& r);
    template<class Y>
      static ProxyHandle checkedCast(const ProxyHandle<Y>& r,
                                     const ::Ice::Context& c);
     // ...
  };
}
```

Methods on Proxy Handles

- A checked cast has the same function for proxies as a C++ **`dynamic_cast`** has for pointers:
  - It allows you to assign a base proxy to a derived proxy.
    - If the base proxy's actual run-time type is compatible with the derived proxy's static type, the assignment succeeds and, after the assignment, the derived proxy denotes the same object as the base proxy.
    - Otherwise, if the base proxy's runtime type is incompatible with the derived proxy's static type, the derived proxy is set to null.

```
BasePrx base = ...; // Initialize base proxy
DerivedPrx derived;
derived = DerivedPrx::checkedCast(base);
if (derived) {
  // Base has run-time type Derived,
  // use derived...
} else {
  // Base has some other, unrelated type
}
```

  - A **`checkedCast`** typically results in a remote message to the server.

Methods on Proxy Handles

- Unchecked cast
  - In some cases, it is known that an object supports a more derived interface than the static type of its proxy. For such cases, you can use an unchecked down-cast:

```
namespace IceInternal {
  template<typename T>
    class ProxyHandle : public IceUtil::HandleBase<T> {
    public:
    template<class Y>
       static ProxyHandle uncheckedCast(const ProxyHandle<Y>& r);
    // ...
  };
}
```

  - An **`uncheckedCast`** provides a down-cast without consulting the server as to the actual run-time type of the object, for example:

```
BasePrx base = ...; // Initialize to point at a Derived
DerivedPrx derived;
derived = DerivedPrx::uncheckedCast(base);
// Use derived...
```

Methods on Proxy Handles

- Stream insertion and stringification
  - For convenience, proxy handles also support insertion of a proxy into a stream, for example:

```
Ice::ObjectPrx p = ...;
cout << p << endl;
```

  - This code is equivalent to writing:

```
Ice::ObjectPrx p = ...;
cout << p->ice_toString() << endl;
```

  - Either code prints the stringified proxy. You could also achieve the same thing by writing:

```
Ice::ObjectPrx p = ...;
cout << communicator->proxyToString(p) << endl;
```

  - The advantage of using the **`ice_toString`** member function instead of **`proxyToString`** is that you do not need to have the communicator available at the point of call.

Using Proxy Methods

- The base proxy class **`ObjectPrx`** supports a variety of methods for customizing a proxy.
- Since proxies are immutable, each of these “factory methods” returns a copy of the original proxy that contains the desired modification.
  - For example, you can obtain a proxy configured with a ten second timeout as shown below:

```
Ice::ObjectPrx proxy = communicator->stringToProxy(...);
proxy = proxy->ice_timeout(10000);
```

  - A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy.
  - With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a down-cast after using a factory method.

```
Ice::ObjectPrx base = communicator->stringToProxy(...);
HelloPrx hello = HelloPrx::checkedCast(base);
hello = hello->ice_timeout(10000); # Type is preserved
hello->sayHello();
```

  - The only exceptions are the factory methods **`ice_facet`** and **`ice_identity`**.
    - Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison

- Proxy handles also support comparison.
  - Specifically, the following operators are supported:

```
operator==
operator!=
```

  - These operators permit you to compare proxies for equality and inequality. To test whether a proxy is null, use a comparison with the literal 0, for example:

```
if (proxy == 0)
  // It's a nil proxy
else
  // It's a non-nil proxy
```

Object Identity and Proxy Comparison

- operator<
- operator<=
- operator>
- operator>=
  - Proxies support comparison.
    - This allows you to place proxies into STL containers such as maps or sorted lists.
- Boolean comparison
  - Proxies have a conversion operator to bool.
    - The operator returns true if a proxy is not null, and false otherwise.

```
BasePrx base = ...;
if (base)
  // It's a non-nil proxy
else
  // It's a nil proxy
```

Object Identity and Proxy Comparison

- Proxy comparison uses all of the information in a proxy for the comparison.
  - Not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same.
  - Comparison with == and != tests for proxy identity, not object identity. A common mistake is to write code along the following lines:

```
Ice::ObjectPrx p1 = ...; // Get a proxy...
Ice::ObjectPrx p2 = ...; // Get another proxy...
if (p1 != p2) {
  // p1 and p2 denote different objects // WRONG!
} else {
  // p1 and p2 denote the same object // Correct
}
```

- Even though p1 and p2 differ, they may denote the same Ice object.
  - This can happen because, for example, both p1 and p2 embed the same object identity, but each use a different protocol to contact the target object.
- If two proxies compare equal with ==, we know that the two proxies denote the same object (because they are identical in all respects);
- If two proxies compare unequal with ==, we know absolutely nothing: the proxies may or may not denote the same object.

Object Identity and Proxy Comparison

- To compare the object identities of two proxies, you can use helper functions in the Ice namespace:

```
namespace Ice {
  bool proxyIdentityLess(const ObjectPrx&,const ObjectPrx&);
  bool proxyIdentityEqual(const ObjectPrx&,const ObjectPrx&);
  bool proxyIdentityAndFacetLess(const ObjectPrx&, const ObjectPrx&);
  bool proxyIdentityAndFacetEqual(const ObjectPrx&,const ObjectPrx&);
}
```

- The **`proxyIdentityEqual`** function returns true if the object identities embedded in two proxies are the same and ignores other information in the proxies, such as facet and transport information.
  - To include the facet name in the comparison, use **`proxyIdentityAndFacetEqual`** instead.
- The **`proxyIdentityLess`** function establishes a total ordering on proxies.
  - It is provided mainly so you can use object identity comparison with STL sorted containers.

Mapping for Operations

- For each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy handle.

```
module Filesystem {
  interface Node {
    idempotent string name();
  };
  // ...
};
```

- The proxy class for the Node interface, tidied up to remove irrelevant detail, is as follows:

```
namespace IceProxy {
  namespace Filesystem {
    class Node : virtual public IceProxy::Ice::Object {
    public:
      std::string name();
      // ...
    };
  typedef IceInternal::ProxyHandle<Node> NodePrx;
  // ...
  }
  // ...
}
```

Mapping for Operations

- The name operation returns a value of type string. Given a proxy to an object of type Node, the client can invoke the operation as follows:

```
NodePrx node = ...; // Initialize proxy
string name = node->name(); // Get name via RPC
```

- The proxy handle overloads operator-> to forward method calls to the underlying proxy class instance which, in turn, sends the operation invocation to the server, waits until the operation is complete, and then unmarshals the return value and returns it to the caller.
- Because the return value is of type string, it is safe to ignore the return value.
  - The following code contains no memory leak:

```
NodePrx node = ...; // Initialize proxy
node->name();       // Useless, but no leak
```

- This is true for all mapped Slice types: you can safely ignore the return value of an operation, no matter what its type - return values are always returned by value.
  - If you ignore the return value, no memory leak occurs because the destructor of the returned value takes care of deallocating memory as needed.

Normal and `idempotent` Operations

- You can add an idempotent qualifier to a Slice operation. As far as the signature for the corresponding proxy methods is concerned, idempotent has no effect.
- Consider the following interface:

```
interface Example {
  string op1();
  idempotent string op2();
  idempotent void op3(string s);
};
```

- The proxy class for this interface looks like this:

```
namespace IceProxy {
  class Example : virtual public IceProxy::Ice::Object {
  public:
    std::string op1();
    std::string op2(); // idempotent
    void op3(const std::string&); // idempotent
    // ...
  };
}
```

- Because idempotent affects an aspect of call dispatch, not interface, it makes sense for the mapping to be unaffected by the idempotent keyword.

Passing In-Parameters

- The parameter passing rules for the C++ mapping are very simple: parameters are passed either by value (for small values) or by const reference (for values that are larger than a machine word).
- Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation (with some caveats)

Passing In-Parameters

```
struct NumberAndString {
  int x;
  string str;
};
sequence<string> StringSeq;
dictionary<long, StringSeq> StringTable;
interface ClientToServer {
  void op1(int i, float f, bool b, string s);
  void op2(NumberAndString ns, StringSeq ss, StringTable st);
  void op3(ClientToServer* proxy);
};
```

```
struct NumberAndString {
  Ice::Int x;
  std::string str;
  // ...
};
typedef std::vector<std::string> StringSeq;
typedef std::map<Ice::Long, StringSeq> StringTable;
namespace IceProxy {
  class ClientToServer : virtual public IceProxy::Ice::Object {
  public:
    void op1(Ice::Int, Ice::Float, bool, const std::string&);
    void op2(const NumberAndString&,
             const StringSeq&,
             const StringTable&);
    void op3(const ClientToServerPrx&);
    // ...
  };
}
```

Passing In-Parameters

- Given a proxy to a **`ClientToServer`** interface, the client code can pass parameters as in the following example:

```
ClientToServerPrx p = ...; // Get proxy...
p->op1(42, 3.14, true, "Hello world!");
                                        // Pass simple literals
int i = 42;
float f = 3.14;
bool b = true;
string s = "Hello world!";
p->op1(i, f, b, s); // Pass simple variables

NumberAndString ns = { 42, "The Answer" };
StringSeq ss;
ss.push_back("Hello world!");
StringTable st;
st[0] = ss;
p->op2(ns, ss, st); // Pass complex variables

p->op3(p); // Pass proxy
```

- You can pass either literals or variables to the various operations. Because everything is passed by value or const reference, there are no memory-management issues to consider.

Passing Out-Parameters

- The C++ mapping passes out-parameters by reference.

```
struct NumberAndString {
  int x;
  string str;
};
sequence<string> StringSeq;
dictionary<long, StringSeq> StringTable;
interface ServerToClient {
  void op1(out int i, out float f, out bool b, out string s);
  void op2(out NumberAndString ns,
           out StringSeq ss,
           out StringTable st);
           void op3(out ServerToClient* proxy);
};
```

```
namespace IceProxy {
  class ServerToClient : virtual public IceProxy::Ice::Object {
  public:
    void op1(Ice::Int&, Ice::Float&, bool&, std::string&);
    void op2(NumberAndString&, StringSeq&, StringTable&);
    void op3(ServerToClientPrx&);
    // ...
  };
}
```

Passing Out-Parameters

- Given a proxy to a ServerToClient interface, the client code can pass parameters as in the following example:

```
ServerToClientPrx p = ...; // Get proxy...
int i;
float f;
bool b;
string s;
p->op1(i, f, b, s);
     // i, f, b, and s contain updated values now
NumberAndString ns;
StringSeq ss;
StringTable st;
p->op2(ns, ss, st);
     // ns, ss, and st contain updated values now
p->op3(p);
     // p has changed now!
```

- Again, there are no surprises in this code: the caller simply passes variables to an operation; once the operation completes, the values of those variables will be set by the server.

Passing Out-Parameters

- It is worth having another look at the final call:
  - `p->op3(p); // Weird, but well-defined`
- Here, p is the proxy that is used to dispatch the call. That same variable p is also passed as an out-parameter to the call, meaning that the server will set its value.
- In general, passing the same parameter as both an input and output parameter is safe
  - The Ice run time will correctly handle all locking and memory-management activities.

Passing Out-Parameters

- Another, somewhat pathological example is the following:

```
sequence<int> Row;
sequence<Row> Matrix;
interface MatrixArithmetic {
  void multiply(Matrix m1,
                Matrix m2,
                out Matrix result);
};
```

- Given a proxy to a MatrixArithmetic interface, the client code could do the following:

```
MatrixArithmeticPrx ma = ...; // Get proxy...
Matrix m1 = ...; // Initialize one matrix
Matrix m2 = ...; // Initialize second matrix
ma->multiply(m1, m2, m1); // !!!
```

- This code is technically legal, in the sense that no memory corruption or locking issues will arise, but it has surprising behavior:
  - Because the same variable m1 is passed as an input parameter as well as an output parameter, the final value of m1 is indeterminate
    - If client and server are collocated in the same address space, the implementation of the operation will overwrite parts of the input matrix m1
  - In general, you should take care when passing the same variable as both an input and output parameter and only do so if the called operation guarantees to be well-behaved in this case.

Chained Invocations

- Consider the following simple interface containing two operations, one to set a value and one to get it:

```
interface Name {
   string getName();
   void setName(string name);
};
```

- Suppose we have two proxies to interfaces of type **`Name`**, p1 and p2, and chain invocations as follows:
  - **`p2->setName(p1->getName());`**
- This works exactly as intended: the value returned by p1 is transferred to p2.
  - There are no memory-management or exception safety issues with this code.

Exception Handling

- Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following interface:

```
exception Tantrum {
 string reason;
};
interface Child {
 void askToCleanUp() throws Tantrum;
};
```

- Slice exceptions are thrown as C++ exceptions, so you can simply enclose one or more operation invocations in a try-catch block:

```
ChildPrx child = ...; // Get proxy...
try {
  child->askToCleanUp(); // Give it a try...
} catch (const Tantrum& t) {
  cout << "The child says: " << t.reason << endl;
}
```

Exception Handling

- Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be dealt with by exception handlers higher in the hierarchy. For example:

```
void run()
{
  ChildPrx child = ...; // Get proxy...
  try {
    child->askToCleanUp(); // Give it a try...
  } catch (const Tantrum& t) {
    cout << "The child says: " << t.reason << endl;
    child->scold(); // Recover from error...
  }
  child->praise(); // Give positive feedback...
}
int main(int argc, char* argv[])
{
  int status = 1;
  try {
    // ...
    run();
    // ...
    status = 0;
  } catch (const Ice::Exception& e) {
    cerr << "Unexpected run-time error: " << e << endl;
  }
  // ...
  return status;
}
```

- This code handles a specific exception of local interest at the point of call and deals with other exceptions generically.
- For efficiency reasons, you should always catch exceptions by const reference.
  - This permits the compiler to avoid calling the exception's copy constructor.

Exception Handling

- Exceptions and Out-Parameters
  - The Ice run time makes no guarantees about the state of out-parameters when an operation throws an exception:
    - The parameter may have still have its original value or may have been changed by the operation's implementation in the target object.
- Exceptions and Return Values
  - For return values, C++ provides the guarantee that a variable receiving the return value of an operation will not be overwritten if an exception is thrown.
    - This guarantee holds only if you do not use the same variable as both an out-parameter and to receive the return value of an invocation.